# Lab Manual

**CS225L – Operating System Lab**
Lab No: **04**
Topic: Shell Scripting

Class: **BSSE**
Semester: **V**
Session: Fall 2025
Instructor: Miss Seemab Khan
Lab Instructor: Miss Seemab Khan

**Air University Islamabad**
**FACULTY OF COMPUTING & ARTIFICAL INTELLIGENCE**
**Department of Creative Technologies**

# Instructions

**<u>Submission</u>:** Use proper naming convention for your submission file. Name the submission file as Lab_NO_DEGREE_ROLLNUM (e.g. Lab_01_BSSE_00000). Submit the file on Google Classroom within the deadline. Failure to submit according to the above format would result in deduction of 10% marks. Submissions on the email will not be accepted.

**<u>Plagiarism</u>:** Plagiarism cases will be dealt with strictly. If found plagiarized, both the involved parties will be awarded zero marks in the assignment, all of the remaining assignments, or even an F grade in the course. Copying from the internet is the easiest way to get caught!

**<u>Deadline</u>:** The deadlines to submit the assignment are hard. Late submission with marks deduction will be accepted according to the course policy shared by the instructor. Correct and timely submission of the assignment is the responsibility of every student; hence no relaxation will be given to anyone.

**<u>Comments</u>:** Comment your code properly. Bonus marks (maximum 10%) will be awarded to well comment code. Write your name and roll number (as a block comment) at the beginning of the solution to each problem.

**<u>Tip</u>:** For timely completion of the assignment, start as early as possible. Furthermore, work smartly - as some of the problems can be solved using smarter logic.

1. Note: Follow the given instructions to the letter, failing to do so will result in a zero.

# Objectives:

In this lab, you will learn:
1. In this lab, you will learn:
2. Shell scripting
3. Variables, operators in Shell
4. Conditional and iterative statements in Shell
5. g++ Compiler Usage
6. Compiling C Basic Program in liunx using g++

# Concepts

## Shell Scripting:

If the shell offers the facility to read its commands from a text file, then its syntax and features may be thought of as a programming language, and such files may be thought of as scripts. So a shell is actually two things:

1. An interface between user and OS.
2. A programming language.

Shell Script is series of commands written in plain text file the shell reads the commands from the file just as it would have typed them into a terminal. The basic advantage of shell scripting includes:

1. Shell script can take input from user, file and output them on screen.
2. Useful to create our own commands.
3. Save lots of time.
4. To automate some task of day today life.
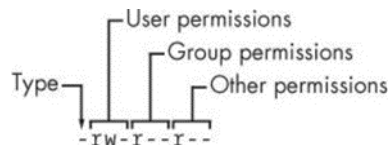5. System Administration part can be also automated.

**EXAMPLES:**

```
# !/bin/bash
#First shell script
clear
echo "Hello World"
```

```
# !/bin/bash
#Script to print user information who currently login, current date and time
clear
echo "Hello $USER"
#echo induce next line at the end
#\c restrict output on same line and used with flag -e
echo -e "Today is \c"; date
#print the line count
echo "Number of lines, total words and characters: "; ls -l | wc
echo "Calendar"
cal
exit 0
```

Use any editor (vi etc) to write shell script. Then save the shell script to a directory and name it intro. Shell scripts don't need a special file extension, so leave the extension blank (or you can add the extension .sh).

## FILE MODES AND PERMISSIONS:

Every Linux file has a set of permissions that determine whether you can read, write, or run the file. Running ls -l displays the permissions. The file's mode represents the file's permissions and some extra information. There are four parts to the mode, as illustrated in figure below:



The first character of the mode is the file type. A dash (-) in this position, as in the example, denotes a regular file, meaning that there's nothing special about the file. This is by far the most common kind of file. Directories are also common and are indicated by a d in the file type slot. The rest of a file's mode contains the permissions, which break down into three sets: user, group, and other, in that order. For example, the rw- characters in the example are the user permissions, the r– characters that follow are the group permissions, and the final r– characters are the other permissions.

| File Mode | Explanation |
|---|---|
| r | Means that the file is readable. Means |
| w | that the file is writable. |
| x | Means that the file is executable (you can run it as a program). |
| - | Means nothing |

The user permissions (the first set) pertain to the user who owns the file. The second set, group permissions are for the file's group (somegroup in the example). Any user in that group can take advantage of these permissions. Everyone else on the system has access according to the third set, the other permissions.

## MODIFYING PERMISSIONS:

To execute a script first we will make it executable. To change permissions, use the chmod command. Only the owner of a file can change the permissions. There are two ways to write the chmod command:

1.  chmod {a,u,g,o}{+,-}{r,w,x}

    {all, user, group, or other}, {read, write, and execute} '+' means add permission and '-' means remove permission.
    For example, to add group (g) and others (o) read (r) permissions to file, you could run

these two commands:

$ chmod g+r file
$ chmod o+r file

Or you could do it all in one shot:

$ chmod go+r file

To remove these permissions, use go-r instead of go+r.

2. chmod ⟨3DigitNumber⟩

Every digit sets permission for the owner(user), group and others as shown in Table 2. To set the permission decipher the number first and then execute the command as:

$ chmod 700 file

| User | | | Group | | | Other | | | 3 Digit No. | Description |
|---|---|---|---|---|---|---|---|---|---|---|
| r | w | x | r | w | X | r | w | x | | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 400 | user: read |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 040 | group: read |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 711 | user: read/write/execute; group, other: execute |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 644 | user: read/write; group, other: read |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 600 | user: read/write; group, other: none |

Execute your script as:

./script-name

## IMPORTANT POINTS:

Following are some important points while writing a script:

1. # is used to write comments in your script
2. Whenever a semicolon (;) is placed between two commands, shell will treat them as separate commands. So if you want to write all the commands in your shell script in one line place a semicolon in between them.
3. A script may start with the line #!/bin/bash to tell your interactive shell that the program which follows should be executed by bash.

## VARIABLES IN SHELL:

In Linux shell there are three types of variables:

1. User defined or shell variables
2. System or Environment variables
3. Parametric variables\

## USER DEFINED OR SHELL VARIABLES:

The shell can store temporary variables, called shell variables, containing the values of text

strings. Shell variables are very useful for keeping track of values in scripts. To assign a value to a shell variable, use the equal sign (=). Here's a simple example:

$$\text{\$ STUFF=blah}$$

The preceding example sets the value of the variable named STUFF to blah. To access this variable, use

$STUFF (for example, try running echo $STUFF).

## Rules for defining variables:

1. Variable name must begin with Alphabetic character or underscore character, followed by one or more Alphanumeric character.
2. Don't put spaces on either side of the equal sign when assigning value to variable.
3. Variables are case-sensitive, just like filename in Linux.
4. You can define NULL variable as

   var=

   var=""

5. Do not use ?,* etc, to name your variable names.

```
# !/ bin / bash
#
# variables in shell script
#
myvar=Hello
echo $myvar
myvar="Yesdear"
echo $myvar
myvar=7+5
echo $myvar
```

## SYSTEM OR ENVIRONMENT VARIABLES:

An environment variable is like a shell variable, but it's not specific to the shell. All processes on UNIX systems have environment variable storage. The main difference between environment and shell variables is that the operating system passes all of your shell's environment variables to programs that the shell runs, whereas shell variables cannot be accessed in the commands that you run. Assign an environment variable with the shell's export command. For example, if you'd like to make the $STUFF shell variable into an environment variable, use the following:

$$\text{\$ STUFF=blah}$$

$$\text{\$ export STUFF}$$

Environment variables are useful because many programs read them for configuration and options. Listed below are some common environment variables:

1. $PATH

PATH is a special environment variable that contains the command path (or path for short). A command path is a list of system directories that the shell searches when trying to locate a command. For example, when you run ls, the shell searches the directories listed in PATH for the ls program. If programs with the same name appear in several directories in the path, the shell runs the first match- ing program. If you run:

$ echo $PATH

You'll see that the path components are separated by colons (:).

For example:

$ echo $PATH

will output /usr/local/bin:/usr/bin:/bin

To tell the shell to look in more places for programs, change the PATH environment variable. For example, by using this command, you can add a directory dir to the beginning of the path so that the shell looks in dir before looking in any of the other PATH directories.

$ PATH=dir:$PATH

Or you can append a directory name to the end of the PATH variable, causing the shell to look in dir last:

$ PATH=$PATH:dir

2.      $HOME

The home directory of the current user.

3.      $IFS

An input field separator; a list of characters that are used to separate words when the shell is reading input, usually space, tab and newline characters.


## PARAMETRIC   VARIABLES:


These variables keep the values of the command line arguments passed to the Scripts. Most shell scripts understand command-line parameters and interact with the commands that they run. These parametric variables passed to the script make the code more flexible. These variables are like any other shell variable like Environment and Shell Variables, except that you cannot change the values of certain ones. Following are some parametric variables and a set of environment variables used to handle parametric variables.

1.      Individual Arguments: $1, $2, ...

$1, $2, and all variables named as positive nonzero integers contain the values of the script parameters, or arguments. For example, say the name of the following script is pshow:

```
# !/ bin / bash
echo First argument: $1
echo Third argument: $3
```

Try running the script as follows to see how it prints the arguments:

$ ./pshow one two three

Output looks like:
First argument: one
Third argument: three

The built-in shell command shift can be used with argument variables to remove the first argument ($1) and advance the rest of the arguments forward. Specifically, $2 becomes $1, $3 becomes $2, and so on. For example, assume that the name of the following script is shiftex:

```
#!/bin/bash
echo Argument: $1
shift
echo Argument: $1
shift
echo Argument: $1
```

Run it like this to see it work:
$ ./shiftex one two three Argument:
one Argument: two Argument: three
As you can see, shiftex prints all three arguments by printing the first, shifting the remaining arguments, and repeating.
The shell maintains a variable called $# that contains the number of items on the command line in addition to the name of the command ($0).

```
# !/ bin / bash
if [ $# -gt 0 ]; then
    echo "Your command line contains $# arguments"
    echo "All arguments displayed using \$* positional parameter"
else
    echo "Your command line contains no arguments"
fi
```

1. Number of Arguments: $#
   The $# variable holds the number of arguments passed to a script and is especially important when running shift in a loop to pick through arguments. When $# is 0, no arguments remain, so $1 is empty.
2. Script Name: $0
   The $0 variable holds the name of the script, and it is useful for generating diagnostic

messages. For example, say your script needs to report an invalid argument that is stored in the $errormsg variable. You can print the diagnostic message with the following line so that the script name appears in the error message:

echo $0: $errormsg

3.      **Process ID: $$**

The $$ variable holds the process ID of the shell.

**INPUT VALUES:**

The read command is used to get a line of input into a variable.

```
# !/bin/bash
#
# Script to read your name from key-board
#
echo "Your first name please :"
read fname
echo "Hello $fname , Lets be friend!"
```

Rules to input variables:

1.      Each argument must be a variable name without the leading "$".
2.      The built in command reads a line of input and separates the line into individual words using the "IFS" inter field separator.
3.      Each word in the line is stored in a variable from left to right.
4.      The first word is stored in the first variable, the second word to the second variable and so on.
5.      If there are fewer variables than words, then all remaining words are then assigned to the last variable.
6.      If you have more variables than words defined, then any excess variables are set to null.

```
# !/bin/bash
#
# Script to read your name from key-board
#
read first middle last
echo "Hello $first $middle $last"
```

**CONDITIONAL STATEMENTS:**

The Bourne shell has special constructs for conditionals, such as

1. If statement
2. If-else statement
3. If-elif statement
4. Case Statement

➢ **IF STATEMENT:**

The basic syntax of if statements is:

```
# !/ bin / bash
if [ conditional expression ]
then
    statement1
    statement2
fi
```

➢ **TESTING CONDITION:**

<div align="center">test command or [ expr ]</div>

are used to see if an expression is true, and if it is true it return zero(0) otherwise returns nonzero for false.

The following script determines whether given number is equal to 100 or not.

```
#!/ bin / bash
#
# Script to see whether argument is positive
count =100
if [ $count -eq 100 ]
then
        echo "Count is 100 "
fi
```

The following script determines whether given argument number is positive using test.

```
#!/bin / bash
#
# Script to see whether argument is positive
if test $1 -gt 0
then
        echo "$1 number is positive"
fi
```

# Mathematical Operators:

Figure given below shows different conditions that we can use to test our expressions. It's important to recognize that the equal sign (=) looks for string equality, not numeric equality. Therefore, [ 1 = 1 ] returns 0 (true), but [ 01 = 1 ] returns false. When working with numbers, use -eq instead of the equal sign: [ 01 -eq 1 ] returns true.

| Mathematical Operator in Shell Script | Meaning | Normal Arithmetical/ Mathematical Statements | But in Shell | |
|---|---|---|---|---|
| | | | For test statement with if command | For [ expr ] statement with if command |
| -eq | is equal to | 5 == 6 | if test 5 -eq 6 | if [ 5 -eq 6 ] |
| -ne | is not equal to | 5 != 6 | if test 5 -ne 6 | if [ 5 -ne 6 ] |
| -lt | is less than | 5 < 6 | if test 5 -lt 6 | if [ 5 -lt 6 ] |
| -le | is less than or equal to | 5 <= 6 | if test 5 -le 6 | if [ 5 -le 6 ] |
| -gt | is greater than | 5 > 6 | if test 5 -gt 6 | if [ 5 -gt 6 ] |
| -ge | is greater than or equal to | 5 >= 6 | if test 5 -ge 6 | if [ 5 -ge 6 ] |

## Logical Operators:

You can invert a test by placing the ! operator before the test arguments. For example, [ !
-f file ] returns true if file is not a regular file. Furthermore, the -a and
-o flags are the logical and and or operators. For example, [ -f file1 -a file2 ].

| Operator | Meaning |
|---|---|
| ! expression | Logical NOT |
| expression1 -a expression2 | Logical AND |
| expression1 -o expression2 | Logical OR |

## String comparisons:

Script checks whether the script's first argument is "hi" where a string comparison is made.

```
#!/bin/bash
#
if [ $1 = hi ]; then
echo "The first argument was hi"
fi
```

There is a slight problem with the condition in preceding example due to a very common mistake:
$1 could be empty, because the user might not enter a parameter. Without a parameter, the test
reads [ = hi ], and the command aborts with an error. You can fix this by enclosing the parameter
in quotes:

if [ "$1" = hi ]; then

| Operator | Meaning |
|---|---|
| string1 = string2 | string1 is equal to string2 |

**Air University Islamabad**
**FACULTY OF COMPUTING & ARTIFICAL INTELLIGENCE**
Department of Creative Technologies

| string1 != string2 | string1 is NOT equal to string2 |
|---|---|
| -n string1 | string1 is NOT NULL and does exist |
| -z string1 | string1 is NULL and does exist |

-z and -n work with the string enclosed within double quotes.

```
# !/ bin / bash
# String Comparisons

string1 =$1
if [ -z "$string1" ] ; then
        echo "NULL STRING"
fi
if [ -n "$string1" ] ; then
        echo "NOT NULL STRING"
fi
```

## IF ELSE STATEMENT:

The basic syntax of if-else statements is:

```
if condition
then
     execute  all  commands upto else
else
    execute  all  commands upto fi
fi
```

Examples using if-else:

```
# !/ bin / bash
#script to see whether  argument  is  positive  or  negative
if [ $# -eq 0 ]
then
        echo "$0 :You  must  supply  one  integer"
        exit 1
fi
if test $1 -gt 0
then
        echo "$1 is  positive number"
else
        echo "$1 is  negative "
fi
```

```
# !/ bin / bash
count =99
if [ $count -eq 100 ]
then
   echo "Count is  100 "
else
   echo "Count is  not  100"
fi
```

## IF-ELIF STATEMENT:

**Air University Islamabad**
**FACULTY OF COMPUTING & ARTIFICAL INTELLIGENCE**
Department of Creative Technologies

The basic syntax of if-elif statements is:

```
if condition
then
        . . . . . . . . . . . .
elif condition
then
        . . . . . . . . . . . .
else
        . . . . . . . . . . . .
fi
```

Example:

```bash
# !/ bin / bash
echo "Is it morning? Please answer yes or no"
read timeofday
if[ $timeofday = "yes" ]; then
        echo "Good morning"
elif [ $timeofday = "no" ]; then
        echo "good afternoon "
else
        echo "Sorry, $timeofday not recognized . Enter yes or no"
        exit 1
fi
exit 0
```

## CASE STATEMENT:

For each case value, you can match a single string or multiple strings with | (hi|hello returns true if $1 equals hi or hello in third example), or you can use the * or ? patterns (what*). To make a default case that catches all possible values other than the case values specified, use a single * as shown by the final case in the examples. The syntax of case statement is:

```
case $variable -name in
pattern 1) command................command ;;
pattern 2) command................command ;;
patternN) command............command ;;
*) command............ command ;;
esac
```

Examples:

**Air University Islamabad**
**FACULTY OF COMPUTING & ARTIFICAL INTELLIGENCE**
**Department of Creative Technologies**

```bash
# ! / bin / bash
echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
yes)  echo "Good Morning"  ;;
no )  echo "Good Afternoon" ;;
y )  echo "Good Morning" ;;
n )  echo "Good Afternoon" ;;
* )  echo "Sorry, answer not recognized" ;;
esac
```

```bash
# ! / bin / bash
case $1 in
bye)
    echo Fine , bye .
    ;;
hi|hello)
    echo Nice to see you.
    ;;
what *)
    echo Whatever.
    ;;
*)
echo 'Huh?'
;;
esac
```

# Arithmetic Operators:

Arithmetic expansion and evaluation is done by placing an integer expression using the following format:

$$((\text{expression}))$$

$$(( n1+n2 ))$$

The shell script below includes some arithmetic operations.

```bash
# !/ bin / bash
clear
echo " hello world"
var1 =10;
var2 =20;
var3=$ (( $var1 + $var2 ));
var4=$(( $var1 - $var2));
var5=$ (( $var1 * $var2 ));
var6=$ (( $var1 / $var2 ));
echo $var 3 ;
echo $var 4 ;
echo $var 5 ;
echo  $var 6 ;
exit
```

# Loops:

There are two kinds of loops in the shell: for and while loops.

## 1. For Loop

The basic syntax of for loop is:

```
for variable_name in { list of values }
do
      execute one for each item in the list until list is not finished
      repeat all statement between do and done
done
```

Examples:

```
#!/bin/bash
for i in 1 2 3 4 5
do
    echo "Welcome $i times"
done
```

```
#!/bin/bash
i=1
for day in Mon Tues Wed Thu
do
      echo "Weekday $((i++))  : $day"
done
```

Sample Output:
```
Weekday 1 : Mon
Weekday 2 : Tue
Weekday 3 : Wed
Weekday 4 : Thu
```

```bash
#!/bin/bash
#define interval in for loop
#the loop execute from 1 to 5
for i in {1..5}
do
    echo "Welcome $i times"
done
```

```bash
#!/bin/bash
#the loop execute from 0 to 10
#with a jump of 2 in each interation
for i in {0..10..2}
do
    echo "Welcome $i times"
done
```

## 2. While Loop

The basic syntax of while loop is:

```bash
#!/bin/bash
while [ condition ]
do
    command_1
    command_2
    . . . . . . . . . . . .
    command_N
done
```

Examples:

```bash
#!/bin/bash
echo "Enter password"
read trythis

while [ "$trythis" != "secret" ]; do
    echo "Sorry, try again"
    read trythis
done
```

```bash
#!/bin/bash
#set n to 1
n=1

#continue until n equals 5
while [ $n -le 5 ]
do
    echo "Welcome $n times."
    n=$(( n+1 ))
done
```

Sample Output:
  Welcome 1 times.
  Welcome 2 times.
  Welcome 3 times.
  Welcome 4 times.
  Welcome 5 times.

**Lab Tasks:**

- Write a Shell program to check the given year is leap year or not
- Write a Shell program to find the factorial of a number
- Write a Shell program to swap the two integers
- Write a Shell program that takes 2 numbers and displays their sum and product.
- Write a Shell program to display your name, registration number, age and date of birth.
- Write a Shell program that an integer input and displays its 10 multiples.